

Advanced Perl

© Bernd Klein

Quote

Griet to Vermeer:

(after seeing her own portrait):

You looked inside me

Regular Expressions

Pattern Matching

Pattern Matching

- A *pattern* is a sequence of characters to be searched for in a character string
 - */pattern/*
- Match operators
 - `=~`: tests whether a pattern is matched
 - `!~`: tests whether patterns is not matched

Definition

A regular expression is a pattern to be matched against a string.

This matching either succeeds or fails.

Sometimes, this is all we are interested in.

At other times, we want to do some changes to the string if the pattern matches in a certain way.

grep, sed, and the like

Regular expressions can be found in many UNIX/Linux commands: grep, sed, awk, vi, ed, emacs and in most shells.

The regular expressions of Perl are a semantic superset of all these commands and programs, i.e. any regular expression of one of those tools can be expressed in Perl, but not necessarily in the same syntax

Simple Word Matching

The simplest regexp is just a word, or more generally, a string of characters.

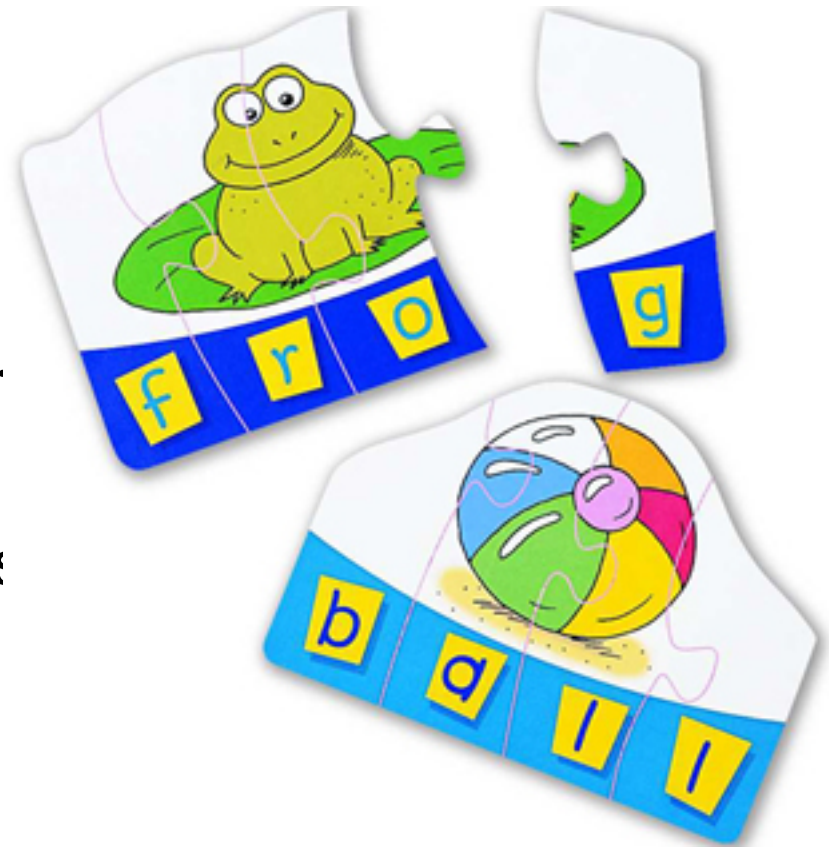
A regexp consisting of a word matches any string that contains that word.

```
$the_simpsons =~ /homer/
```

Simple Word Matching

The simplest regexp is just a word, or more generally, a string of characters.

A regexp consisting of a word matches any string that contains that word.



```
$the_simpsons =~ /homer/
```


Simple Word Matching, Example

```
$the_simpsons = "homer marge bart lisa maggie";  
if ($the_simpsons =~ /homer/) {  
    print "homer is a member of the Simpsons  
family!\n";  
} else {  
    print "homer is not a member of the Simpsons  
family!\n";  
}
```

This script can be improved by including a variable into the regular expression instead of “homer”.

Improved Script

```
print "Name? ";
chomp ($name = <STDIN>);

$the_simpsons = "homer marge bart lisa maggie";
if ($the_simpsons =~ /$name/) {
    print "$name is a member of the Simpsons
family!\n";
} else {
    print "$name is not a member of the Simpsons
family!\n";
}
```

The next disturbing thing: Names have to be typed in lower case.

We can make the test case-insensitive by appending an `i` to `/ $name /`

Case-insensitive Matching with /i

Now we make the previous example case-insensitive:

```
print "Name? ";
chomp ($name = <STDIN>);

$the_simpsons = "homer marge bart lisa maggie";
if ($the_simpsons =~ /$name/i) {
    print "$name is a member of the Simpsons
family!\n";
} else {
    print "$name is not a member of the Simpsons
family!\n";
}
```

Substitute Operator

The substitute operator replaces the part of a string that matches the regular expression with another given string.

It is similar to the s command of the sed tool of UNIX/Linux.

```
s /be /have / ;
```

The variable `$_` is matched against the regular expression `be`. If the match is successful, the matching part of the string will be replaced by the replacement string `have`.

Substitute Example

```
$_ = "To be or not to be that is the  
question!";  
s/be/have/;  
print "$_\n";
```



The script above prints the following line:

To **have** or not to **be** that is the question!

What if the programmer had in mind to change both “be”s with “have”?

No problem! The option g makes sure that every – not just the first from left – will be replaced.

Substitute Example with “g”

```
$_ = "To be or not to be that is the question!";  
s/be/have/g;  
print "$_\n";
```

The extended script prints now the following line:

```
To have or not to have that is the question!
```

Metacharacters

<code>\</code>	Quote the next metacharacter
<code>^</code>	Match the beginning of the line
<code>.</code>	Match any character (except newline)
<code>\$</code>	Match the end of the line (or before newline at the end)
<code> </code>	Alternation (matches either the expression preceding or following <code> </code>)
<code>()</code>	Grouping regular expressions
<code>[]</code>	Character class
<code>{ }</code>	Range of Occurences

Quantifier Metacharacters

`*` Match 0 or more times

`+` Match 1 or more times

`?` Match 1 or 0 times

`{n}` Match exactly n times

`{n,}` Match at least n times

`{n,m}` Match at least n but not more than m times

Examples with Metacharacters

Whats wrong with the following expressions?

```
"1+ 1 = 3" =~ /1+1/;
```

```
"/data/proj1/gtk/abc.odf" =~ /gtk/abc.odf/;
```

The correct version:

```
"1+ 1 = 3" =~ /1\+1/;
```

```
"/data/proj1/gtk/abc.odf" =~ /gtk\./abc\./odf/;
```

Emulation of UNIX/Linux grep

The following script is a simple emulation of the `grep` command.

If the script is saved as `simple_grep`, it can be invoked with:

```
> simple_grep homer addresses.txt
```

```
#!/usr/bin/perl
```

```
$regexp = shift;  
while (<>) {  
    print if /$regexp/;  
}
```

Using Character Classes

A character class make it possible to use a set of possible characters rather than just a single character.

They are denoted by brackets [. . .]

```
/[bcefhor]at/;
```

This pattern can match the words:

bat, cat, eat, fat, hat, oat, rat

Changing it a little bit, we can match all the English 3-letter words ending with at.

```
/[a-z]at/i;
```

Special Characters in Character Classes

- \ escape character
- denotes a range of characters, e.g. a-z
exception: in the first or last position it stands for a regular character
-] end marker for the character class
- ^ in the first position denotes the negated character class, e.g.
[^0-9] stands for “every character except a digit”
- \$ is special, as it is used with scalar variables

Other characters are not special within the bracket context!

Fat, Cat, Rat, Eat, Oat and the Like

**Just relax
before it
gets
tricky.**

Tricky Example

```
$x = 'force' ;
```

```
/[$x]at/ ;
```

matches: fat, oat, rat,
cat, eat

```
/[\$x]at/ ;
```

matches: \$at or xat

```
/[\\$x]at/ ;
```

matches: \at, fat, oat,
rat, cat, eat

Predefined Character Classes

<code>\w</code>	Match a "word" character (alphanumeric plus "_")
<code>\W</code>	Match a non-"word" character
<code>\s</code>	Match a whitespace character
<code>\S</code>	Match a non-whitespace character
<code>\d</code>	Match a digit character
<code>\D</code>	Match a non-digit character

Freedom of choice

The pattern `/homer/` is a shortcut for `m/homer/`

If you use `m` as a prefix, you don't have to stick to a pair of slashes as delimiters.

This means, that the following expressions are equivalent:

`m(homer)`
`m<homer>`
`m{homer}`
`m[homer]`

Even more astonishing:

`m,homer,`
`m!homer!`
`m^homer^`
and many others

Hierarchical Matching

Parts of a regular expression can be grouped by enclosing them in parenthesis (grouping metacharacters).

```
/(friend|hard|war)ship/  
matches 'friendship', 'hardship', 'warship'
```

Matching year numbers by prefixing the century:

```
/(19|20|)\d\d/;
```

Note the `|` after the 20 to match year numbers without century prefix!

Character Classes and Alternation

The alternation metacharacter `|` can be used to match different possible words or character strings.

Example:

```
/homer|marge|bart|lisa|maggie/
```

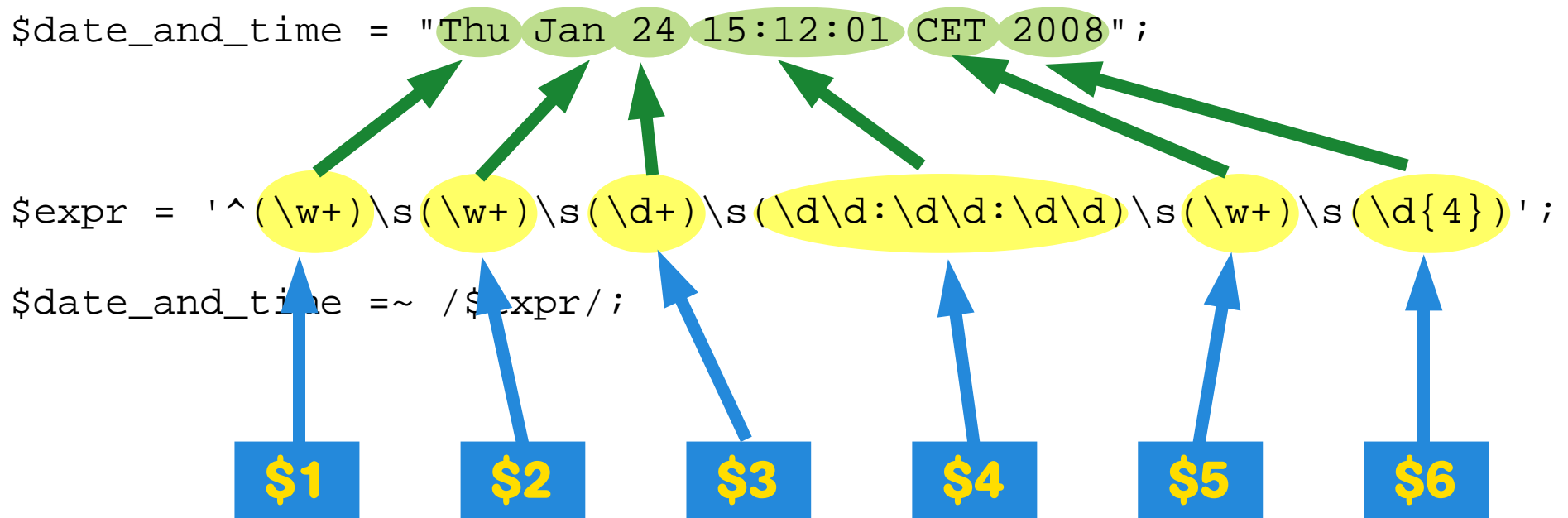
The Character Classes are like special cases of the alternation for single characters.

```
/a|b|c/ matches the same strings as /[abc]/
```

There is a difference between the patterns `/ho|hom|homer/` and `/homer|hom|ho/`.

Extracting Matched Substrings

Besides grouping regular expressions, grouping parenthesis allows the extraction of the substrings of a string that matched the group.



`$^N` is set to the result of the last completed capture group.

Backreferences

Backreferences (`\1`, `\2`, ...) are matching variables (like `$1`, `$2`, ...) that can be used inside a regular expression.

With the help of a backreference earlier matched substrings can be matched again.

Task:

Check for doubled words in a text, like the typical “the the”

Solution:

```
/\b(\w+)\s\1\b/
```


Check with:

```
grep.pl '\b(\w+)\s\1\b' 1984.txt
```

Extended Regular Expressions

The regular expressions can be extended with a pair of parentheses with a question mark at the first position within the parentheses.

The character after the question mark determines the function of the extension.

(?  . . .)

 a character to define the function, e.g. ':',
'=', '!' etc.)

Non-capturing Groupings

Groupings with parentheses () have two functions:

- 1) group regexp elements as a single unit, and
- 2) capture (extract) substrings that matched the regexp in the grouping.

(?:regexp) allows to group a regexp as a single unit without extracting a substring or setting a matching variable e.g. \$1.

```
# $1-$4 are set, but we only need $1 and $4  
/([+-]? \*(\d+(\.\d*)?|\.\d+)([eE][+-]?\d+)?) /;
```

```
# $1 = whole number, $2 = exponent  
/([+-]? \*(?:\d+(?:\.\d*)?|\.\d+)(?:[eE]([+-]?\d+))?) /;
```

Extensions

<code>(?#text)</code>	A comment. The “text” is ignored.
<code>(?:...)</code>	This groups things like “(. . .)” but doesn’t make backreferences.
<code>(?=...)</code>	A zero-width positive lookahead assertion. For example, <code>/w+(?=\t)/</code> matches a word followed by a tab, without including the tab in <code>\$&</code> .
<code>(?!...)</code>	A zero-width negative lookahead assertion. For example, <code>/foo(?!bar)/</code> matches any occurrence of “foo” that isn’t followed by “bar”.
<code>(?<=...)</code>	A zero-width positive lookbehind assertion. For example, <code>/(?<=bad)boy/</code> matches the word boy that follows bad, without including bad in <code>\$&</code> . This only works for fixed-width lookbehind.
<code>(?<!=...)</code>	A zero-width negative lookbehind assertion. For example, <code>/(?<!=bad)boy/</code> matches any occurrence of “boy” that doesn’t follow “bad”. This only works for fixed-width lookbehind.
<code>(?>...)</code>	Matches the substring that the standalone pattern would match if anchored at the given position.

Further Extensions

`(?(condition)yes-pattern|no-pattern)`

`(?(condition)yes-pattern)`

A pattern is determined by a condition. The condition should be either an integer, which is "true" if the pair of parentheses corresponding to the integer has matched, or a lookahead, lookbehind, or evaluate, zero-width assertion. The no-pattern will be used to match if the condition was not meant, but it is also optional.

`(?imsx-imsx)`

One or more embedded pattern-match modifiers. Modifiers are switched off if they follow a - (dash).

Modifier	Meaning
i	Do case-insensitive pattern matching
m	Treat string as multiple lines
s	Treat string as single line
x	Use extended regular expressions

Pattern Code Expression

The pattern code expression is like a regular code expression, except that the result of the code evaluation is treated as a regular expression and matched immediately.

```
$repetitions = 5;  
$char = 'a';  
$x = 'aaaaabb';  
# the x in the next line means "use  
extended expressions"  
$x =~ /(??{$char x $repetitions})/x;
```

It's like

```
$x =~ /(aaaaa)/x;
```

Fibonacci Sequence

```
$s0 = 0; $s1 = 1; # initial conditions
$x = "1101010010001000001";
print "It is a Fibonacci sequence\n"
    if $x =~ /^1                # match an initial '1'
        (
            (??{'0' x $s0}) # match $s0 of '0'
            1                # and then a '1'
            (?{
                $largest = $s0; # largest seq so far
                $s2 = $s1 + $s0; # compute next term
                $s0 = $s1;      # in Fibonacci sequence
                $s1 = $s2;
            })
        )+ # repeat as needed
    $      # that is all there is
/x;

print "Largest sequence matched was $largest\n";
```

Previous regexp without /x

```
/^1( (??{ '0'x$s0} ) 1 ( ? { $largest=$s0; $s2=$s1+  
$s0$s0=$s1; $s1=$s2; } ) ) +$/;
```

Fibonacci Sequence, even trickier

```
$s0 = ''; $s1 = '0';    # initial conditions
$x = "1101010010001000001";
print "It is a Fibonacci sequence\n"
    if $x =~ /^1          # match an initial '1'
        (?:
            ((?{ $s0 } )) # match some '0'
            1             # and then a '1'
            (?{ $s0 = $s1; $s1 .= $^N; })
        )+               # repeat as needed
    $                    # that is all there is
/x;

printf "Largest sequence matched was %d\n", length($s1)-
length($s0);
```

Exercise

- 1) Write an expression to check for words consisting of two identical parts.
- 2) Write a programm to find the palindrome words in a text



Solution

1) Words consisting of two identical parts:

```
/\b(\w+)\1\b/
```

2) Palindrome words in a text

```
while (<>) {  
    print $1 , "\n" if /\b((\w+)\w?(?){reverse  
$2} ) )\b/ig;  
}
```

All About Strings

The Perl String

One of the most useful features of Perl is its text processing and manipulation abilities. At the heart of its powerful string manipulation facilities.

Perl's string functions can be used to (among other things)

- print and format strings,
- split and join string values,
- alter string case, and
- perform regular expression searches.

Finding a Substring with `index`

```
$position = index($haystack, $needle);
```

Perl locates the first occurrence of the string `$needle` in the string `$haystack` and returns an integer, indicating the location of its first character.

If `$needle` can't be located inside `$haystack` a `-1` will be returned. `0` if `$needle` matches first letter of `$haystack` and `(n - 1)` for the `n`th position in `$haystack`.

Example: index

```
$dr_seuss="shuffle, duffle, muzzle, muff";
```

```
$p=index($dr_seuss, "uff");
```

```
print "$p\n";
```

**2**

```
$p=index($dr_seuss, "uff", $p+1);
```

```
print "$p\n";
```

**10**

```
$p=index($dr_seuss, "uff", $p+1);
```

```
print "$p\n";
```

**26**

Finding the last occurrence of the substring with **rindex()**

```
$p = rindex($dr_seuss, "uffle");
```

```
print "$p\n";
```

**10**

```
$p = rindex($dr_seuss, "uffle", $p + 1);
```

```
print "$p\n";
```

**2**

```
$p = rindex($dr_seuss, "uffle", $p + 1);
```

```
print "$p\n";
```

**-1**

Manipulating a Substring with substr

substr takes two or three arguments:

- a string value (the one where a substring will be cut off)
- the initial position of the substring
- the length of the substring

```
$ulysses= "like mad and yes I said yes I will Yes."
```

```
$phrase = substr($ulysses,17,10);
```

```
print "$phrase\n";
```



"I said yes"

```
$phrase = substr($ulysses,13,10);
```

```
print "$phrase\n";
```



"yes I said"

```
$u = $ulysses;
```

```
$phrase = substr($u,rindex($u,"yes"),-6);
```

```
print "$phrase\n";
```



"yes I will"

Cut and Paste with substr ()

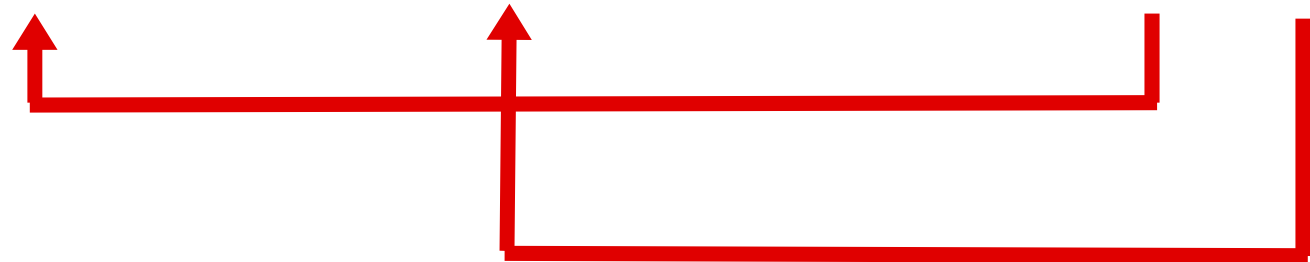
```
$cover = "yes she said yes she will Yes." ;  
$needle = "she" ;  
$prefix = "like mad and " ;  
substr($cover,  
        index($cover, $needle),  
        length($needle))="I" ;  
print "$cover\n" ;  
substr($cover,  
        index($cover, $needle),  
        length($needle))="I" ;  
print "$cover\n" ;  
$ulysses = $prefix . $cover ;  
print "$ulysses\n" ;
```



“like mad and yes I said yes I will Yes.”

Formatting Output with `printf`

```
$x = "she";  
printf "yes %s said yes %s will Yes.\n", $x, $x;
```



The format, the first string following the `printf` command, contains the conversions:

(%) sign followed by a letter or letters.

There should be the same number of items in the following list as there are conversions; if these don't match up, it won't work correctly.

Printing numbers with `printf`

The easiest way to print a number with `printf` is `%g`, the “general” numeric conversion.

```
printf "%g, %g\n", 4*3, 1/3;
```

`%d` prints decimal integers, truncated (not rounded) if necessary:

```
printf "Number of items: %d\n", 7.35;
```




Number of items: 7

```
printf "The answer to life ... %3d\n",  
21.03*1.9998;
```



The answer to life ... 42

Floating-Point Conversion

<code>printf "%8f\n", 21/34;</code>		0.617647
<code>printf "%8.0f\n", 21/34;</code>		1
<code>printf "%8.2f\n", 21/34;</code>		0.62
<code>printf "%8.5f\n", 21/34;</code>		0.61765
<code>printf "%8.9f\n", 21/34;</code>		0.617647059

General format of %f:

%<number of digits>.<digits after the decimal point>f

Exercise

Write a program that asks the user to enter a list of strings on separate lines.

Find the longest line (`$max_line`) and print each string right-justified in a `${max_line}`-column.

What has to be changed to print the lines left-justified?

Solution

```
print "Enter your lines, press Ctrl-D or Ctrl-Z  
to finish!\n";  
chomp(my @lines = <STDIN>);  
print "1234567890" x 7, "12345\n";    # ruler line  
  
# find longest line  
$max_line = 0;  
foreach (@lines) {  
    if ($max_line < length($_)) {  
        $max_line = length($_);  
    }  
}  
foreach (@lines) {  
    printf "%${max_line}s\n", $_;  
}
```

The Split Function

The `split()` function is used to split a string into smaller sections.

The substrings are placed into an array

A string can be split on

- a single character,
- a group of characters or
a regular expression (a pattern).

You can also specify how many pieces to split the string into.

Example

```
my $data = 'George Bush,1600 Pennsylvania Avenue  
NW,Washington,DC 20500';  
  
my @parts = split(',', $data);  
  
foreach my $x (@parts) {  
    print "$x\n";  
}
```

This script produces the following output:

```
George Bush  
1600 Pennsylvania Avenue NW  
Washington  
DC 20500
```

Example with Regular Expressions

```
my $data = 'surname: Bush, prename: George,  
profession: president';  
  
my @parts = split(/,* * \w*:/, $data);  
foreach my $x (@parts) {  
    print "$x\n";  
}
```

This script produces the following output:

```
Bush  
George  
president
```

The Join Function

The `join()` function is inverse to `split()`. It can be used to (re)join the array elements (a list of values), i.e. the values will be glued together with a glue string between the parts.

`join()` takes two arguments:

- 1) the glue, a scalar variable to use as a separator
- 2) an array

It returns a string that contains the elements of the array “glued together” by the given separator

The Join Function: Example

```
my $data = 'George Bush,1600 Pennsylvania Avenue  
NW,Washington,DC 20500';  
my @parts = split(',', $data);  
my $data2 = join(", ", @parts);  
if ($data == $data2) {  
    print "equal";  
} else {  
    print "not equal";  
}
```

Exercises

- 1) Write a regular expression that matches:
 - a) any number of backslashes followed by an arbitrary number of asterisks
 - b) at least three but not more than five consecutive copies of whatever is contained in a variable `$container`.
 - c) at least five characters, including newline
 - d) the same word written two or more times in a row with possibly varying number of whitespaces between them. A word is a nonempty sequence of nowhitespace characters.
- 2) Write a script to print the login name and the real name of each user in `/etc/passwd`

Answers

1) Write a regular expression that matches:

- a) any number of backslashes followed by an arbitrary number of asterisks

```
/\\*\\**/
```

- b) at least three but not more than five consecutive copies of whatever is contained in a variable \$container.

```
/($container){3,5}/
```

- c) any five characters, including newline

```
/(.|\n){5,}/
```

- d) the same word written two or more times in a row with possibly varying number of whitespaces between them. A word is a nonempty sequence of nowhitespace characters.

```
/(\\s)(\\S+)(\\s+\\2)+(\\s|$)/
```

2) Write a script to print the login name and the real name of each user in /etc/passwd

Answers

Write a script to print the login name and the real name of each user in `/etc/passwd`:

```
while (<STDIN>) {  
    chomp;  
    ($user, $rname) = (split /:/)[0,4];  
    ($real) = split(/,/, $rname);  
    print "The real name of $user is $real\n";  
}
```

Converting Other Languages to Perl

New Chapter

Converting awk to Perl

Perl is a lot more comprehensive than awk, but most of awk can be found in a similar form in Perl. So Perl is a kind of superset of awk.

Hardly surprising, there is an “awk to Perl”-converter:
`a2p AwkFile.awk`

A converted awk script will usually perform the identical functionality, often with an increase in speed, and without any of awk's built-in limits on line lengths, parameter counts and others.

Example

The simplest awk script is the one just “echoing” the input (file) again.

```
{  
    print $0;  
}
```

The following Perl code has the same functionality:

```
while (<>) {  
    chomp;  
    print $_;  
}
```

Invoking a2p yields the following code

```
#!/usr/bin/perl
eval 'exec /usr/bin/perl -S $0 ${1+"$@"}'
    if $running_under_some_shell;
    # this emulates #! processing on NIH
machines.
    # (remove #! line above if
indigestible)

eval '$'. $1. '$2;' while $ARGV[0] =~ /^([A-Za-z_0-9]+=)(.*)/
&& shift;
    # process any FOO=bar switches

#!/usr/bin/perl

$, = ' ';          # set output field separator
$\ = "\n";         # set output record separator

while (<>) {
    chomp;          # strip record separator

    print $_;
}
```

Exercise

Translate an awk script by using a2p or not, which checks for lines in 1984.txt, in which the words “Big Brother” and “watching” appear, into Perl.



```
/[bB]ig [bB]rother.*watching/      {print $0};
```

The solution

Translate an awk script, which checks for lines in which the words “Big Brother” and “watching” appear, into Perl.

```
/[bB]ig [bB]rother.*watching/      {print $0};
```

The Perl equivalent:

```
while (<>) {  
    chomp;  
    if (/[bB]ig [bB]rother.*watching/) {  
        print $_;  
    }  
}
```

Converting sed Programs to Perl

The sed-to-Perl translator is called s2p.
It works similar to a2ps.

Changing all instances of “Big Brother” into “Little Brother”, can be done with sed by invoking

```
sed "s/Big Brother/Little Brother/g" 1984.txt
```

This script can be transformed into Perl code by invoking:

```
echo "s/Big/Little/g" | s2p -f -
```

This results into 121 lines of code!

References

New Chapter

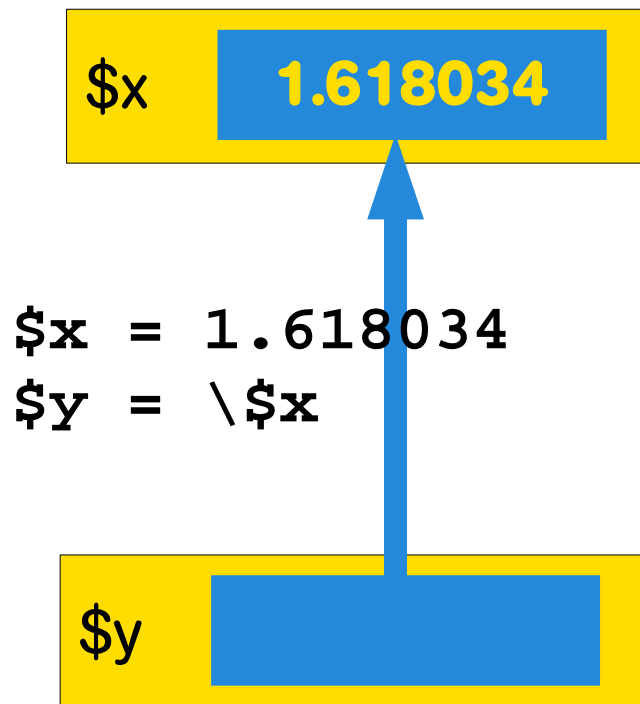
References

References are what the name implies, a reference or a pointer to another object.

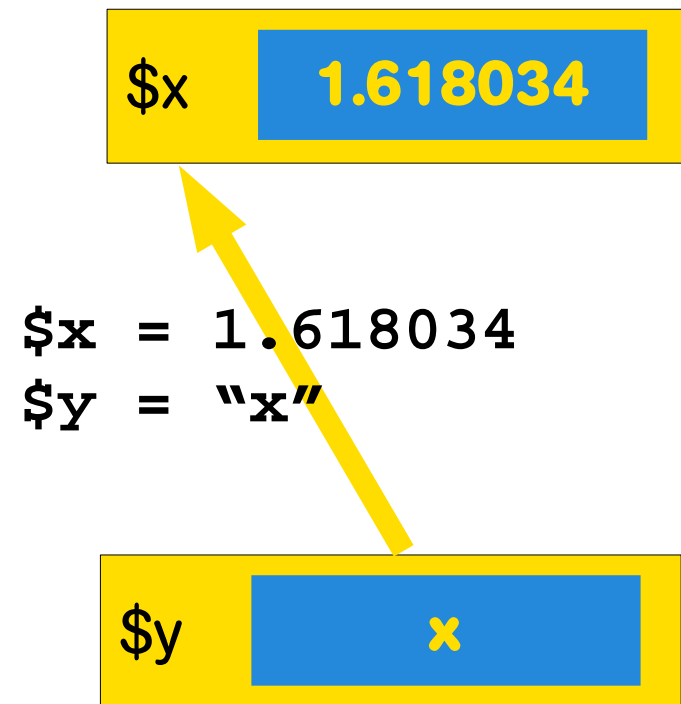
There are symbolic and hard references.

References: Hard and Soft

A hard link



A soft link



Example with References

```
$joyce = "Stately, plump Buck Mulligan";  
$james = \$joyce;  
print "$$james $joyce\n";  
$joyce = "came from the stairhead";  
  
print "$$james $joyce\n";
```

The output of the above script:

```
Stately, plump Buck Mulligan Stately, plump Buck  
Mulligan  
came from the stairhead came from the stairhead
```

Anonymous Data

```
$pi_ref = \3.14159265358979
```

The value of π can only be accessed via the reference `$pi_ref`, as there is no “regular” scalar variable associated to the value.

It's called an anonymous referent.

Anonymous Arrays

A reference to an array can be created directly, i.e. without creating an intervening named array.

```
$some_simpsons = [ 'homer', 'marge', 'bart' ] ;
```

There are two ways to access the elements of an anonymous array:

```
$$some_simpsons[1]  
$some_simpsons->[0]
```

Stepping Through an Anonymous Array

An anonymous array can be addressed with e.g.
`@$some_simpsons`

```
foreach $name (@$some_simpsons) {  
    print "$name\n";  
}
```

Anonymous Hashes

As easy to create as anonymous arrays:

```
$simpsons = { 'Homer' => 'fat and stupid',  
              'Marge' => 'patient and affable',  
              'Bart'  => 'cheeky and lazy',  
              'Lisa'  => 'clever and musical',  
              'Maggie' => 'cute'  
};
```

There are also two ways to access an anonymous hash:

```
$simpsons->{Homer}  
$$simpsons{Marge}
```


Packages

New Chapter

Packages



Packages in Perl

If more than one programmer are working together, the risk is rising that the same names are used for variables, functions etc.

The same is true, if you merge together two or more Perl programmes.

Packages are used in Perl to partition the global namespace.

Every global identifier (variables, functions, file and directory handles, and formats) consists of two parts: its package name separated by a double colon (::) from the actual identifier.

Setting the Default Package Prefix

`package` is a compile-time declaration that sets the default package prefix for unqualified global identifiers.

If no package declaration is given, a default package called `main` is valid.

A package declaration is valid until

- the end of the current scope (a brace-enclosed block, file, or `eval`)
- a subsequent package statement in the same scope.

Example

```
$x = 1;
{
    package Pack1;
    $x = 2;
    print "x in Pack1: $x\n";
}
print "x outside Pack1: $x\n";
package Pack2;
$x = 3;
print "x in Pack2: $x\n";
print "main::x in Pack2: $main::x\n";
```



\$main::x



\$Pack1::x



\$main::x



\$Pack2::x

Output:

```
x in Pack1: 2
x outside
Pack1: 1
x in Pack2: 3
main::x in
Pack2: 1
```

Forced to be in `main`

The identifiers `STDIN`, `STDOUT`,
`STDERR`, `ARGV`, `ARGVOUT`, `ENV`,
`INC`, `SIG` and the punctuation names like
`$_` and `$.` are forced to be in `main`.

Modules

Principally, a module is just a package defined in a file with the same name and the extension `.pm`.

A module name should be capitalized unless it's a pragma.

Perl is distributed with a large number of modules.

They can be found e.g. in `/usr/lib/perl5/5.8.8/` (name and path depending on the used Perl distribution)

Perl modules are typically included in a program with:

```
use MODULE LIST;
```

**It's not necessary to write
`MODULE.pm`**

or:

```
use MODULE;
```

use or require?

A module can be included into a program either with `use` or `require`.

The `use` statement does a preload of a MODULE at compile time and then an import of the requested symbols.

`require` does not necessarily pull in the module at compile time.

`use` can have a list of strings naming entities, which need to be imported from the module.

example:

```
use module qw(const1, const2, func1, func2);
```


Example with Math::Complex

```
use Math::Complex;
$z = Math::Complex->make(5, 6);
$z = cplx(5, 6);      # shorter than previous line
$t = 4 - 3*i + $z;    # do standard complex math
print "$t\n";         # prints 9+3i

print sqrt(-2), "\n"; # prints i

print sin($z);
```

Creating Modules

A Module “as basic as basic can be”:

```
package HelloWorld;

sub say_hello {
    return "Hello, World!";
}

1;
```

The name in the package declaration must match the name of the file, i.e. `HelloWorld.pm`

The module has to return a `true` value. That's why the line with `“1 ;”` is included.

Script Using the Module

A script using our simple module:

```
#!/usr/bin/perl
use HelloWorld;
print HelloWorld::say_hello()."\n";
```

A module can be pulled into a script with Perl's use command. use automatically searches all the paths in the array @INC until it finds a file with the specified name and a .pm extension.

An additional path can be included with

```
BEGIN { push @INC, '/my/dir' }      # or
BEGIN { unshift @INC, '/my/dir' }  # or
use lib '/my/dir';
```

my and our in the package

```
package HelloWorld;

use strict;

our $VERSION=0.1;
my $hello_string="Hello World!";

sub say_hello {
    return $hello_string;
}

1;
```

Invoking the Changed Package

```
#!/usr/bin/perl

use strict;
use HelloWorld;
print "Using version: $HelloWorld::VERSION
of HelloWorld.\n";
print HelloWorld::say_hello()."\n";
```

As `$hello_string` is a “my” variable in the package, so it cannot be accessed from the script, not even by fully qualifying it!

The script can be called now with `use strict 0.1` but called with “`use strict 0.2`” will not load our module, because the version number would have to be equal or higher than 0.2.

Invoking the Changed Package

In the previous example, we have to address variable `$VERSION` and the function `say_hello()` with a fully qualified notation.

If we address them without “HelloWorld::” as a prefix, we get the error messages:

```
Global symbol "$VERSION" requires explicit package name at  
HelloWorld.pl line 7.  
Execution of HelloWorld.pl aborted due to compilation errors.
```

The following slides provide a package with correct exporting.

The Complete HelloWorld Module

```
package HelloWorld;
use strict;
require Exporter; ←
our @ISA = qw(Exporter);
our @EXPORT_OK = qw(say_hello);
our $VERSION=0.1;
my $hello_string="Hello World!";
sub say_hello {
    return $hello_string;
}
1;
```

```
#!/usr/bin/perl
```

```
use strict;
use HelloWorld qw(say_hello);
print "Using version: $HelloWorld::VERSION of
HelloWorld.\n";
print say_hello()."\n";
```

Defines, that we want to utilize the export facilities of the standard Perl module Exporter.

The import routine in Exporter will be called each time our module is used.

The Complete HelloWorld Module

```
package HelloWorld;
use strict;
require Exporter;
our @ISA = qw(Exporter); ←
our @EXPORT_OK = qw(say_hello);
our $VERSION=0.1;
my $hello_string="Hello World!";
sub say_hello {
    return $hello_string;
}
1;
```

The module will inherit functions from the exporter module.

It will be the name of the import module when the module will be used

```
#!/usr/bin/perl
```

```
use strict;
use HelloWorld qw(say_hello);
print "Using version: $HelloWorld::VERSION of
HelloWorld.\n";
print say_hello()."\n";
```


The Complete HelloWorld Module

```
package HelloWorld;
use strict;
require Exporter;
our @ISA = qw(Exporter);
our @EXPORT_OK = qw(say_hello); ←
our $VERSION=0.1;
my $hello_string="Hello World!";
sub say_hello {
    return $hello_string;
}
1;
```

A list of variable names that will be exportable to programs using this module.

```
#!/usr/bin/perl
```

```
use strict;
use HelloWorld qw(say_hello);
print "Using version: $HelloWorld::VERSION of
HelloWorld.\n";
print say_hello()."\n";
```

The Complete HelloWorld Module

```
package HelloWorld;
use strict;
require Exporter;
our @ISA = qw(Exporter);
our @EXPORT_OK = qw(say_hello);
our $VERSION=0.1;
my $hello_string="Hello World!";
sub say_hello {
    return $hello_string;
}
1;
```

The use statement contains a list of the module symbols that should be imported into the namespace of the script.

But it will be still possible to access \$VERSION with its fully qualified name.

```
#!/usr/bin/perl
```

```
use strict;
```

```
use HelloWorld qw(say_hello);
```

```
print "Using version: $HelloWorld::VERSION of
HelloWorld.\n";
```

```
print say_hello()."\n";
```

Automatic Export

The following version of our module exports `say_hello` automatically, so that it doesn't have to be in the symbol list of the `use` statement of the calling script.

```
package HelloWorld;

use strict;
require Exporter;
our @ISA = qw(Exporter);
our @EXPORT = qw(say_hello);
our $VERSION=0.1;
my $hello_string="Hello World!";
sub say_hello {
    return $hello_string;
}
1;
```

Exercise

Add a method in the HelloWorld module which prints a personal greeting, i.e. the method is called with a parameter, the name of the one to be greeted.

Solution, .pm

```
package HelloWorld_personal;  
  
use strict;  
require Exporter;  
our @ISA = qw(Exporter);  
our @EXPORT = qw(say_hello say_name);  
our $VERSION=0.2;  
my $hello_string="Hello World!";  
  
sub say_hello {  
    return $hello_string;  
}  
sub say_name {  
    return $_[0];  
}  
1;
```

Solution: .pl

```
#!/usr/bin/perl
```

```
use strict;
```

```
use HelloWorld_personal;
```

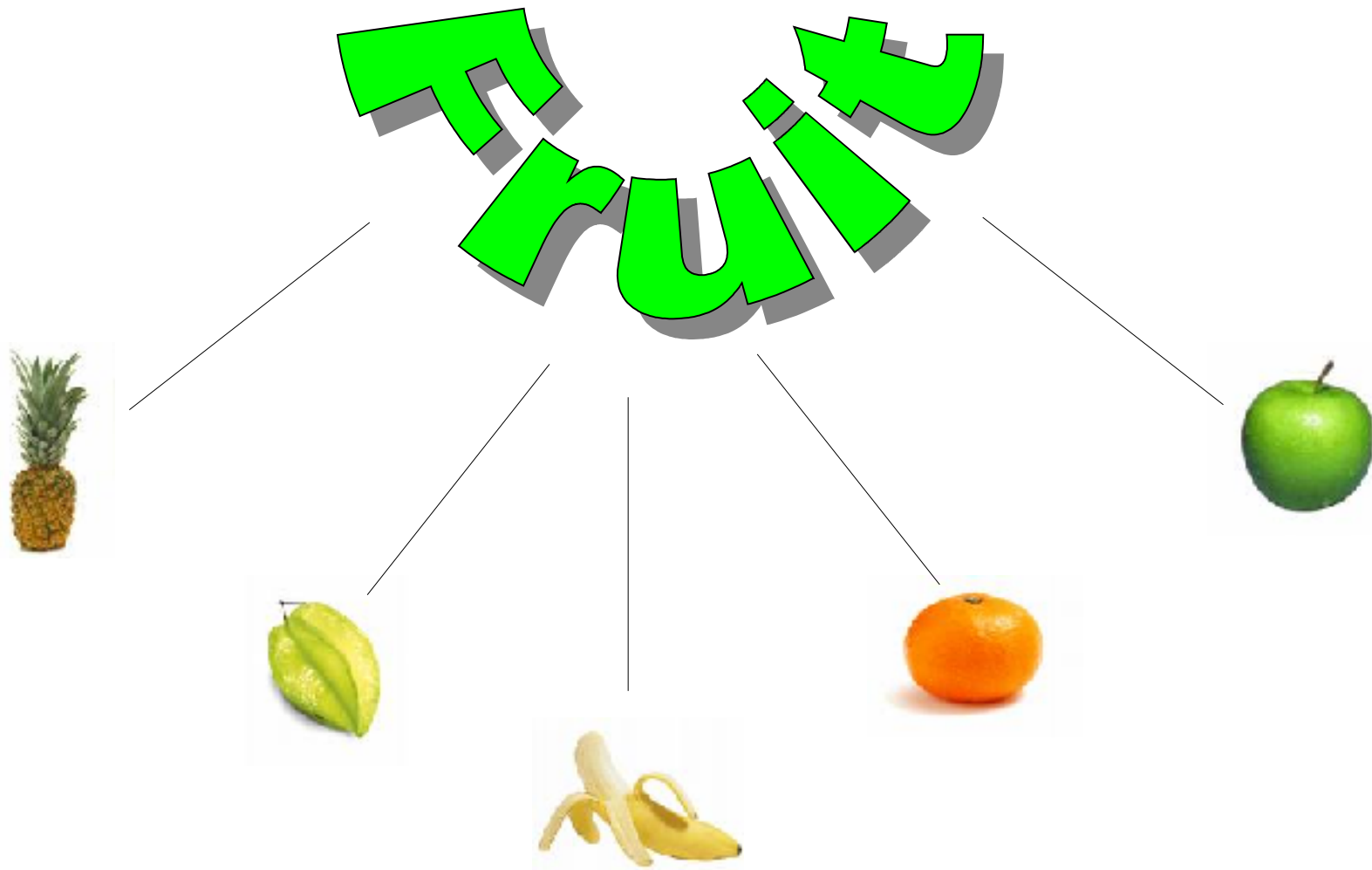
```
print "Using version:
```

```
$HelloWorld_personal::VERSION of HelloWorld.\n";
```

```
print say_hello()."\n";
```

```
print say_name("Stephen")."\n";
```


Objects and Classes



Advantage

In programs like Java you are forced to use objects to write programmes.

Perl allows you to use as many or as few object-oriented techniques as you want.

Nevertheless Perl offers the whole arsenal of object oriented programming: classes, objects, single and multiple inheritance, instance methods and class methods, constructors and destructors, operator overloading and so on

Criticism

Edsger W. Dijkstra wrote: “... *what society overwhelmingly asks for is snake oil. Of course, the snake oil has the most impressive names –otherwise you would be selling nothing– like “Structured Analysis and Design”, “Software Engineering”, “Maturity Models”, “Management Information Systems”, “Integrated Project Support Environments” “Object Orientation” and “Business Process Re-engineering” (the latter three being known as IPSE, OO and BPR, respectively).*”

Alexander Stepanov suggested that OOP provides a mathematically-limited viewpoint and called it, “almost as much of a hoax as Artificial Intelligence”

“I used OOP when working on the Lisp Machine window systems, and I disagree with the usual view that it is a superior way to program.” (Richard Stallman, 1995)

Object Oriented Programming in Perl

Object-oriented programming (OOP) is a programming paradigm that uses "objects" and their interactions to design applications and computer programs.

It is based on several techniques:

- **encapsulation**
- **modularity**
- **polymorphism**
- **inheritance**
- **abstraction**

Perltoot -

Tom's object-oriented tutorial for perl

“Object-oriented programming is a big seller these days. Some managers would rather have objects than sliced bread. Why is that?”

“An object is nothing but a way of tucking away complex behaviours into a neat little easy-to-use bundle. (This is what professors call abstraction.) Smart people who have nothing to do but sit around for weeks on end figuring out really hard problems make these nifty objects that even regular people can use. (This is what professors call software reuse.) Users (well, programmers) can play with this little bundle all they want, but they aren't to open it up and mess with the insides. Just like an expensive piece of hardware, the contract says that you void the warranty if you muck with the cover. So don't do that.”

Some Simple Definitions

1. An **object** is simply a reference that happens to know which class it belongs to.
In other words: An object is a variable that belongs to a class.
2. A **class** is simply a package that happens to provide methods to deal with object references.
3. A **method** is a function associated with a class or object. In other words: It's simply a subroutine that expects an object reference (or a package name, for class methods) as the first argument.

Steps to Create a Class

Build a package

```
package Person;
```

**Subroutines
create methods**

```
sub new {  
    my $class = shift;  
    my $self = { @_ };  
    bless $self, $class;  
}
```

**Bless a referent
to create an object**

```
sub get_first {  
    my $self = shift;  
    # return the first name  
    $self->{'first'};  
}
```

Using Classes

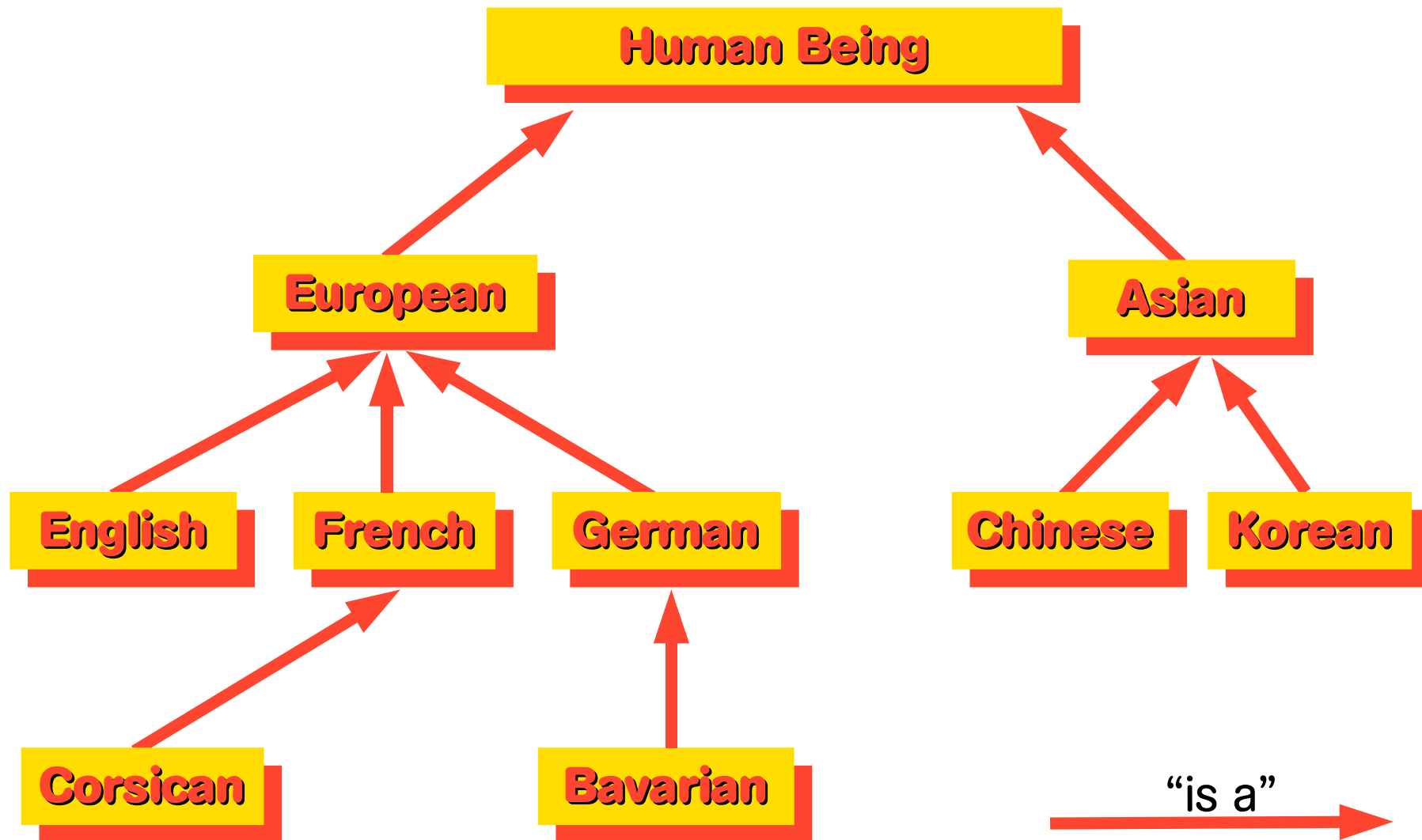
```
use Person;

$Buck = new Person( 'first' => 'Malachi',
                    'last' => 'Mulligan');
$Molly = new Person( 'first' => 'Molly',
                     'last' => 'Tweedy');

print $Buck->get_last(). "'s first name: \n";
print $Buck->get_first(), "\n";
print $Molly->get_last() . "'s first name: \n";
print $Molly->get_first(), "\n";

$Molly->set_last('Bloom');
print $Molly->get_first() . " has married now \n";
print "and is called " . $Molly->get_last(). "\n";
```

Inheritance



Employee --> Person

Every Employee is a Person.

We will extend our example with a class “Employee”.

It inherits the methods of Person.

Employee Class

```
@ISA = qw(Person);  
my $BasePhone = "089/898-";  
  
# method for setting and getting the values  
sub internal_phone {  
    my $self = shift;  
    if (@_) { $self->{'internal_phone'} = shift }  
    $self->{'internal_phone'};  
}  
  
sub company_phone {  
    my $self = shift;  
    if (@_) { $BasePhone = shift }  
    $BasePhone . $self->{'internal_phone'};  
}  
  
1;
```

Improvement to Method `company_phone`

```
sub company_phone {  
    my $self = shift;  
    if (@_) { $BasePhone = shift }  
    if ($self->{'internal_phone'}) {  
        $BasePhone . $self->{'internal_phone'};  
    } else {  
        $BasePhone . "0";  
    }  
}
```

Method Overloading

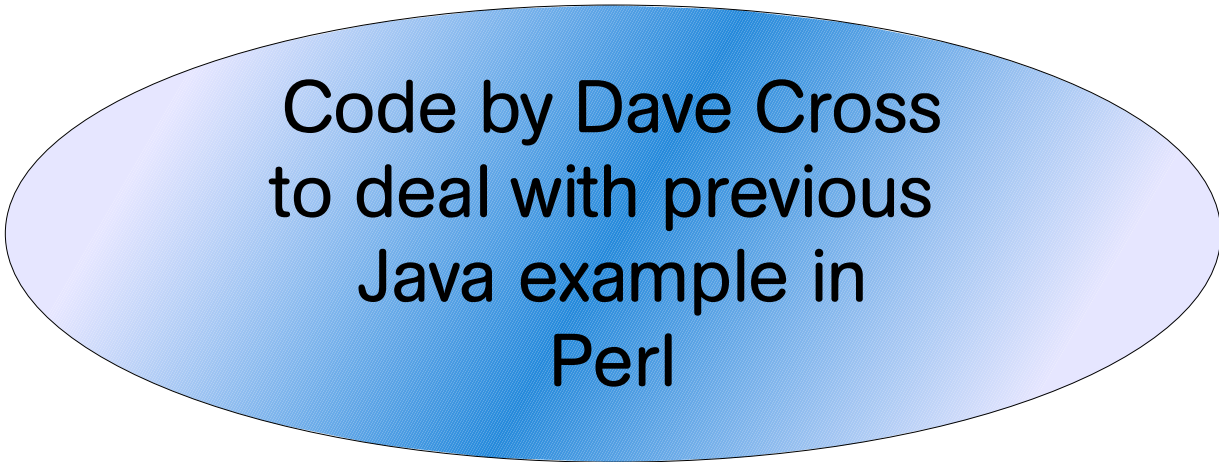
Method overloading is a feature found in various object oriented programming languages such as C#, C++ and Java that allows the creation of several functions with the same name which differ from each other in terms of the type of the input and the type of the output of the function.

Java Example:

```
public Fraction(int num, int den);  
public Fraction(Fraction F);  
public Fraction();
```

Method Overloading, cont.

We need only one method in Perl to deal with the three Java methods. Method overloading as shown in the example is not needed in Perl.



Code by Dave Cross
to deal with previous
Java example in
Perl

Operator Overloading

Formal definition:

Operator overloading is a special case of polymorphism in which some or all operators like `+`, `=`, `>=` aso. have different implementations depending on the types of their arguments.

Some overloadings are defined by the language others are implemented by the programmer.

Overloading offers a way to write

`a + b * c`

instead of

`add(a, multiply(b, c))`

Example: Vector Class

A vector class with arbitrary dimension.

A method to add to vectors and a print function.

Usage looks like this:

```
#!/usr/bin/perl
use Vector;

$v1 = new Vector(1, 2, 3, 4, 5);
$v2 = new Vector(3, 4, 5, 2, 1);

$res = $v1 + $v2;
$res->print;
```

Vector Class: add method

```
sub add {  
    my ($X, $Y, $class);  
    my ($arg) = @_;  
  
    if (ref $arg eq 'Vector') {  
        ($X, $Y) = @_  
    } else {  
        ($class, $X, $Y) = @_  
    }  
    my @sum;  
    for (my $counter=0; $counter < @$X; $counter++) {  
        push @sum, $X->[$counter] + $Y->[$counter];  
    }  
    return Vector->new(@sum);  
}
```

$$\begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix} + \begin{pmatrix} 3 \\ 4 \\ 5 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 1+3 \\ 2+4 \\ 3+5 \\ 4+2 \\ 5+1 \end{pmatrix}$$

Vector Class: print vector

```
sub print {  
    my $self = shift;  
    print "[";  
    for (my $counter=0;  
         $counter < @$self;  
         $counter++) {  
        print $self->[$counter] . " ";  
    }  
    print "]\n";  
}
```


The Complete Vector Class



The Package

Script Using
the Package

Exercise

Enrich the vector class with two methods:
sub to subtract to vectors and a
scalar multiplication.

$$\begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix} - \begin{pmatrix} 3 \\ 4 \\ 5 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 1-3 \\ 2-4 \\ 3-5 \\ 4-2 \\ 5-1 \end{pmatrix}$$

$$5 * \begin{pmatrix} 3 \\ 4 \\ 5 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 5*3 \\ 5*4 \\ 5*5 \\ 5*2 \\ 5*1 \end{pmatrix}$$

Solution: the Extended Vector Class



The Package

Script Using
the Package

Overriding Methods

Let's create a special class of employees

They don't like their direct phone number to be published.

Manager Class

```
package Manager;  
@ISA = qw(Employee);  
  
sub internal_phone {  
    my $self = shift;  
    "hidden"  
}  
sub set_phone {  
    my $self = shift;  
    "hidden";  
}  
sub get_phone {  
    my $self = shift;  
    "hidden";  
}  
1;
```

The Manager class overrides the methods `internal_phone()`, `set_phone()` and `get_phone()` of Employee.

Process Management

New Chapter

Process Management

A Perl program can launch new processes, i.e. child processes.

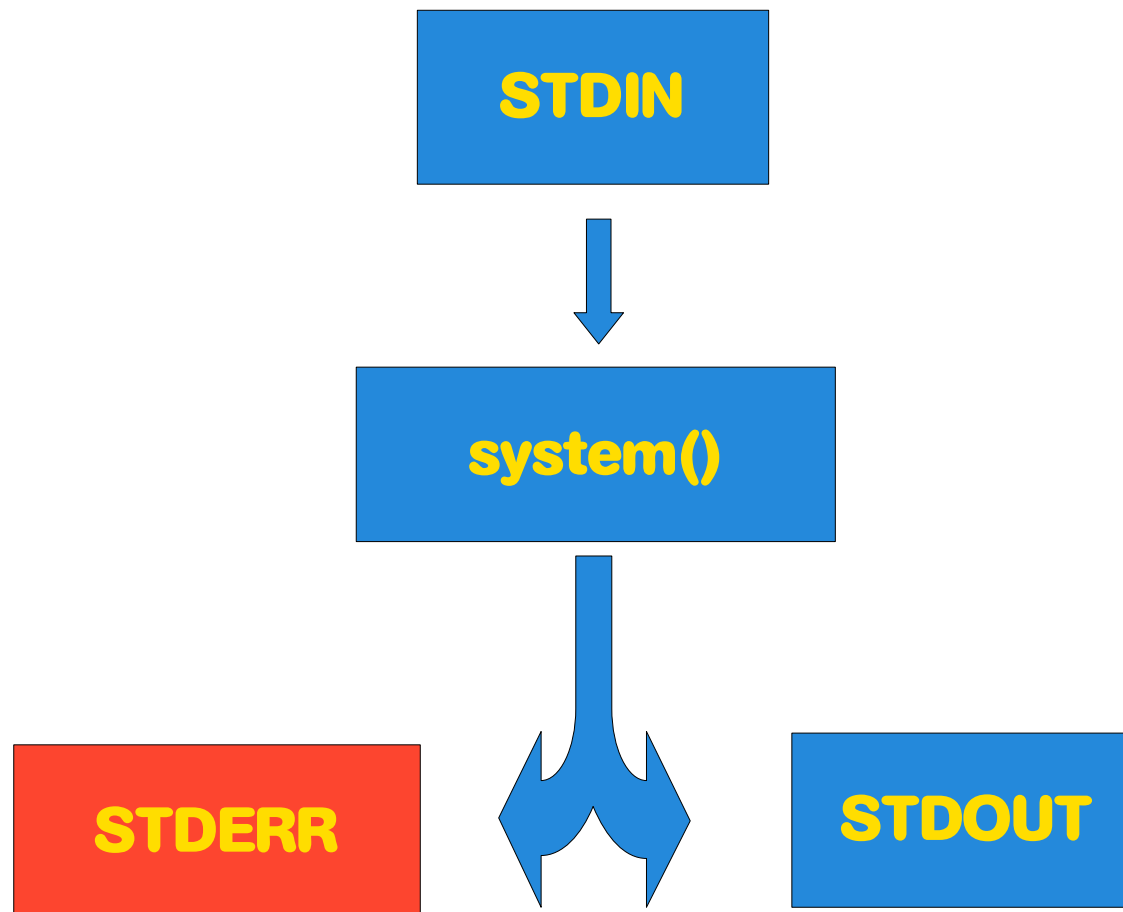
There is more than one way to do so.

The easiest way to launch a new process is to use the `system` function.

Example:

```
system( "date" );
```

Input and output of system()



Changing the Standard Output

```
system("date >date.txt") && die "problems with date()";
```

Inheriting

A child process inherits from its parent: e.g. the current umask, current directory, and the user ID.

All environment variables are inherited by the child.

These variables are typically altered by the `csh setenv` command or the corresponding assignment and export by the `/bin/sh` shell.

Within Perl you can examine and alter environment variables through a hash called `%ENV`.

Each key of this hash corresponds to a name of an environment variable.

This hash contains (if not changed in the script) the environment handed to Perl by the parent shell; altering the hash affects the environment used by Perl and by its child processes, but not the parent.

Print the Environment Variables

A simple script to look at the environment variables:

```
foreach $key (%ENV) {  
    print "$key=$ENV{$key}\n";  
}
```

Background Processes

While a child process, started with `system()` is running, the Perl script will have to wait until it finishes, regardless how long it takes.

A shell command can be launched with an ampersand at the end of the command line. Causing the process to be run in the background.

```
system "some_command_name &"
```

The exec Function

The syntax and semantics of the exec function is nearly the same as system function, except for one thing: The system function creates a child process, which is performing the requested action while Perl has to wait for it to finish.

The exec function causes the Perl process itself to perform the requested action.

Closure

A **closure** is a function that is evaluated in an environment containing one or more bound variables. When called, the function can access these variables. The explicit use of closures is associated with functional programming and with languages such as ML and Lisp. Constructs such as objects in other languages can also be modeled with closures.

Closure: Example

```
sub make_counter {  
    my $start = shift;  
    return sub { $start++ }  
}
```

```
my $from_ten = make_counter(10);  
my $from_three = make_counter(3);
```

```
print $from_ten->()."\n";           # 10  
print $from_ten->()."\n";           # 11  
print $from_three->()."\n";         # 3  
print $from_ten->()."\n";           # 12  
print $from_three->()."\n";         # 4
```